On a Microcode Compiler Toward a Table Driven Firmware Generator (Mathematical Methods in Software Science and Engineering)

FUSAOKA, AKIRA; MIKAMI, KAZUYOSHI

数理解析研究所講究録
363 巻 1979 年 100-129

The Mathematical Methods
in Software Science and Engineering

ON A MICROCODE COMPILER
TOWARD A TABLE DRIVEN FIRMWARE GENERATOR

Akira Fusaoka Kazuyoshi Mikami

Central Research Laboratory
Mitubishi Electric Corporation
Amagasaki-City, Japan 661

ABSTRUCT

This paper describes a PL/1 like language compiler called FGS which generates an efficient microcode of MELCOM-COSMO 500 computer directly. The principal part of this microcode compiler is the resource allocation algorithm in which a two phase allocation procedure is introduced in order to achieve high object code quality for inhomogenious micro-architecture of COSMO 500 computer.

A consideration to attain a machine independent model of microcode compilers is also discussed for supporting a broad class of computers.

INTRODUCTION

Recent decreasing of a memory cost has brought an economic possibility for a microcode implementation of systems and user programs so that the efficient running of the systems is attained. However, because of very low level architecture of micro machine, the support software such as a higher-level language with microprogramming skills are required for program productivity.

There have been many attempts to describe a microprogram by a higher-level language, but the almost all of the works have emphasized on the design methodology for the language that suits to description of microprograms. From a standpoint of a microcode implementation of user programs, however, a microcode compiler for widely used languages such as PL/1, Fortran and LISP are more significant. The first system of a microcode compiler is introduced by C.J.Tan in 1978 which generates a IBM370/145 microcode from a PL/1 source program. (1)

We have developed yet another microcode compiler called FGS (Firmware Generator System) which directly translates a source program written in PL/1 like language into a microcode of MELCOM-COSMO 500 computer. (2)

As the essential requirement for a microcode compiler is the efficiency in the execution speeds and the size of object codes, it is inevitable to use a lot of machine-specific optimization rules in order to produce an efficient code. This leads to the system to be machine-dependent. Especially, a machine structure is radically different in the micro-architecture level, so that it is a very complicated problem to develop a machine independent firmware generator. One of the serious drawbacks of FGS is, in fact, the lack of a portability for target machines.

This paper introduces an overview of FGS and its extentional works to attain a machine independent model of microcode compiler which allows the generation of a microprogram for a broad class of machines.

OVERVIEW OF FGS

Source language

As a source language of FGS, we select a simplified version of PL/S: an IBM dialect of PL/1 for system description. Some luxurious features such as dynamic allocation are eliminated for the simplicity and efficiency of an object program. A sample program written in this source language is given in Figure 1.

SOURCE LISTING

```
/*   TEST PROGRAM NO-TS03   */
MAIN: PROC(MATX,ANS1);
      DCL MATX(10) MM FIXED,
          ANS1     MM FIXED,
          I        MM FIXED;
      ANS1=0;
      DO I=1 TO 10;
      ANS1=ANS1+MATX(I);
      END;
      RETURNA;
      PEND;
```

Figure 1. A sample source program

Structure of a compiler

As shown in Figure 2, the current FGS consists of three phases of processing. The first phase comprises a single pass translator which transforms a source program into a text in an intermediate form. A memory allocation and a high-level, machine independent optimization are also performed in this phase.

The 2nd phase is the essential part of the compiler which contains the resource allocation of the actual micro machine into a program space.
In order to simplify an allocation procedure, a text in the intermediate form is segmented into the straight line program part called a section. A register and function allocation is performed on the basis of this notion of the section.

A text generated in this 2nd phase contains scattered sequences of micro-operations so that the reorganization of these set of micro-operations along with the micro-instruction format is required. This procedure, called a code consolidation, is performed in the 3rd phase.
Some microprogram optimization techniques are also used in a code consolidation to attain an efficient micro-program.

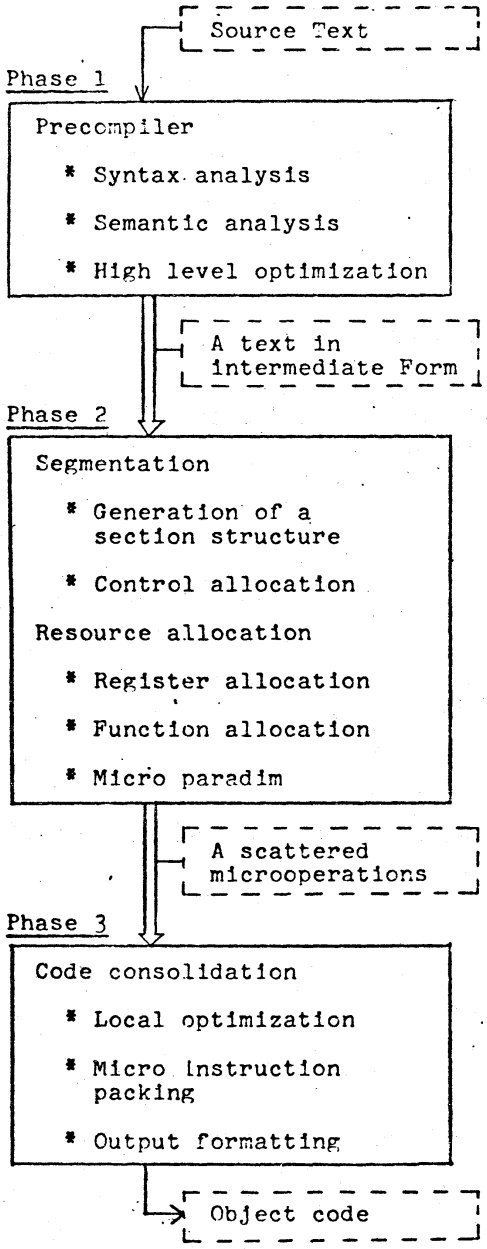

Figure 2. Structure of

A microcode compiler FGS

Intermediate form

In the initial phase of the compiler, a source program is translated into a sequence of the intermediate forms each of which in turn corresponds to a microinstruction of a target machine. The sequence of these forms retains the control structure. The intermeciate forms are divided into two types: the control type and the arithmetic type.

The control type contains the intermediate forms correspoinding to a GOTO, CALL or RETURN statement. The arithmetic type contains the triple form which represents a data calculation or an address calculation.

The control structures of a source program such as a DO statement and a GOTO statement play an important role to the optimized procedure for register allocation. Therefore, the sentential markers for program structure are attached to the intermediate forms. Figure 3 illustrates a part of source program and its corresponding intermediate forms.

To provede a compact representation of the best possible code for each of the possible status settings, a 'microparadigm' is provided for each intermediate form. A microparadigm is a coding skeleton which genetates a concrete microprogram by instanciation of the variable. A sample micro-paradigm is given in Figure 4.

<Source Program>

```
DO  I  =  1  TO  15 ;
    SUM  =  SUM  +  I ;
END ;
```

<Intermediate Form>

```
      asw1  '1'  -> ODi                    [LI]
Lsk   asw2  ODsum + ODi -> ODsum           [LB]
      asw2  ODi + '1'  -> ODi              [LM]
      if    ODi - '15' ≦ 0 Lsk Lfk         [LC]
Lfk   ...
```

Figure 3. An Example of Intermediate Form

```
form      adl   const -> adr
```

```
ADl (const,adr) transfer body;
  constructor body;
    construct a relative address
    on adr;

        CALL CONST (const,adr)

    add a base address

        V(1,SAA4) + adr -> Y;

  end;
```

```
CONST (const,reg) selector body;

  case of

    constructor body;
      const=0: if reg=Ci, Cj then

      reg  <- 0;

                                  else

      0 ->  reg;

    end;

    constructor body;
      1≦const≦15:
        if reg=Ci, Cj, Ck, Cl then

      reg  <-  X'000K';

                                  else

      MD   <-  X'000K';

      reg  <-  MD;

    end;
```

Figure 4. An Example of Micro Paradim

Section structure

The register assignment process in compiler is generally divided into local and global phases.
In order to decide a restricted context for local assignment, an intermediate text is broken into computational "section" whose relationship may be represented by a derected graph that illustrates the flow of control through the program.
Each section consists of a sequence of intermediate forms, only the first of which may be branched to, and only the last on which contains a branch.

The register allocation algorithm of FGS is broken into two stages:

(1) A local assignment in which a register for each variable is determined from the internal information of the section which a variable belongs to.

(2) A global assignment in which the result of local assignment is arranged and coordinated from the boundry condition of neighboring sections.

As shown in Figure 5, a section is represented by a table which contains

(1) Intemediate text

(2) A variable-information at each program point (e.g. reference or assignment)

(3) Span of the variable that represents a distance from the program point of current use of a variable to a point of next use

(4) Control flow information of a section

Section structure

Section name	A List of Section Variables
Pointers to Backward section	Input Condition
Sequence of Intermediate Form	Variable Occurence Table
Pointers to Forward section	Output Condition

An Example of a Section structure

section four	a	b	c	d	e	f	t1
P3	-	C1	-	C2	-	C3	-
asw1 1 -> a	α	1	0	3	0	0	0
asw1 2 -> b	2	α	0	2	0	0	0
asw1 3 -> c	1	2	α	1	0	0	0
asw2 a * d -> t1	β	1	2	β	0	0	α
asw2 t1 + b -> t1	2	β	1	0	0	0	ω
asw2 t1 - c -> e	1	0	β	0	α	0	β
asw2 a + c -> f	β	0	β	0	1	α	0
if e = 0 l1 l2	0	0	0	0	β	0	0
P5 P6	-	C3	C1	C2	-	C5	-

α : assign

β : reference

ω : assign & reference

Ci: register class

n : distance

Figure 5. Section Structure

Code consolidation

In the final phase, a machine dependent optimization is performed in order to enhance the quality of the object code. We have used the following strategies.

(1) Detecting a sequence of meaningless instructions and deleting them.

(2) Combining a set of scattered simple instructions into a single more complex instruction.

In order to detect concurrently executable microinstructions, we have divided the given microprogram into blocks which consist of a sequence of a non-branch microinstruction. Concurrency analysis is performed in three steps. The first step scans symbolic code in each block to determine the variable dependency. The second step scans symbolic code to find a candidate instruction for code consolidation by checking the following field conflict conditions

(1) Does the instruction consist of only auxiliary function?

(2) Does the instruction consist of only test function?

The third step chooses an instruction that consists of only main function and examines whether that instruction and the previously found candidate instruction can be consolidated into a single instruction. Figure 6 illustrates a sample of code consolidation.

After this optimizations, the resulted microprogram is converted into the symbolic external form which is acceptaple by the target machine. Figure 7 shows the resulted micro-program for a MELCOM 500 computer.

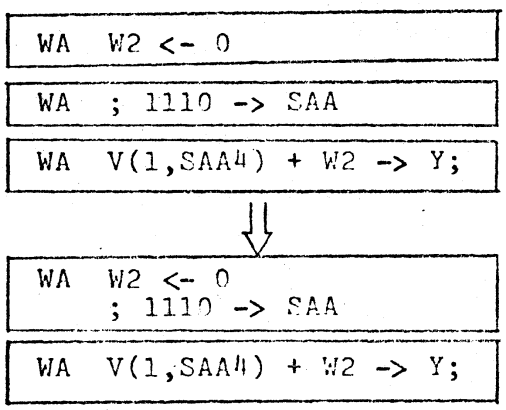

Figure 6. An Example of Code Consolidation.

```
            BEGIN     MAIN
            EXT       DIVDER
C30         DC        X'001E'
C40         DC        X'0028'
C50         DC        X'0032'
ZZRIDT      DS        14
ZZRIIT      DS        6
            EJECT
MAIN        WA        V1          <-          X'000A'
            BA        V2B0        <-          X'00'
            BA        V2B1        <-          X'14'
            WA        V1          ->          W2
            WA        TR          <- 0
            WA        W2          ->          MD                    ;
                      1110->SAA
            WA        V(1,SAA4)   +           TR          ->Y
            MEM       MM(TR) <-MD       ;
                      TR          +           0           ->X       ;
                      MEM.SYNC.
            WA        V2          ->          MD
            WA        V1          +           MD          ->Y
            WA        W2          <-          MD
            WA        TR          <-          X'0001'
            WA        W2          ->          MD                    ;
                      1110->SAA
            WA        V(1,SAA4)   +           TR          ->Y
            MEM       MM(TR) <-MD       ;
                      TR          +           0           ->X       ;
                      MEM.SYNC.
            BA        V1B0        <-          X'00'
            BA        V1B1        <-          X'32'
            WA        V2          ->          MD
            WA        V1          -(+1)       MD          ->Y
            WA        W2          <-          MD
            WA        TR          <-          X'0002'
            WA        W2          ->          MD                    ;
                      1110->SAA
            WA        V(1,SAA4)   +           TR          ->Y
            MEM       MM(TR) <-MD       ;
                      TR          +           0           ->X       ;
                      MEM.SYNC.
```

Figure 7. The resulted microprogram

A TABLE DRIVEN RESOURCE ALLOCATION

In order to attain a machine independent firmware generator, it is necessary to construct a mechnical microprogrammer who accepts the coding rules for a specific target machine and then detailed understanding of micro architecture of the machine.

Such an intelligent compiler, although it is a generator for macroassembler code, (3) is studied by C.W.Fraser. His XGEN system is a code-generator generator which produces a codegenerator for a specific machine from the description of the machine. He uses a production rule to represent a machine structure. This notion of "machine understanding" appears to play a significant role in the design of target machine-independent translator instead of the classical compiler-compiler concept.

Another approach to the machine independent macro generator is given by S.L.Graham. She has utilized a table specification method for machine architecture. (4)

In the following, we discuss about the possibility for a resource allocation algorithm on the basis of a table representation of micro machine architecture. Because the essential part of microcode compiler is a resource allocation, especially register assignment, the machine independent resource allocation algorithm is required.

The many variant searches for efficient algorithms for resource allocation have been performed from the time of the first Fortran compiler for the IBM704.

The main difficulty of this problem in practical situations comes from

(1) The problem to minimize the number of memory trafic (load and store instructions) to evaluate an expression is considered to be NP-complete. (5)

(2) The register usage rules in a micro-architecture are not so homogeneous as discussed in theoretical studies.

Therefore it appears to be an interesting problem to find a technique to organize a resource allocation algorithm automatically from a description of micro-architecture. From the standpoint of resource allocation, only the expression of the machine structure is necessary, though a description of a micro-architecture contains in general the representation of machine structure and its running rule.

A machine structure table

In order to describe a machine structure, we use a table form shown in Figure 8. Each row of the table contains the resource usage rules. Each rule is divided into two parts. The left part is the requirement condition to apply this rule which is expressed in terms of the attributes of variables and a functional expression. The right side of the rule contains the transformation procedure which is a mappine from variables into the registers of the target machine.

This mapping rules contain the following terms.

(1) an ALU function for the corresponding operator in an intermediate form

(2) the class of allowable register for the left operand

(3) the class of allowable register for the right operand

(4) the decision rule of the destination register

(5) the corresponding micro-paradigm

Figure 8 illustrates the table representaion of COSMO 500 micro-architecture.

	ALU operator	Class of X-reg.	Class of Y-reg. C2	C3	C5	C6	KW	KB	KD	Destination register	Micro paradim
16bit ∧ func=add	-(+1)	C1	*		*	*				X-reg or Y-reg	m6
	-(+1)	C2			*	*					
	-(+1)	C3 ∨ C4	*		*						
	-(+1)	C5	*	*							
	-(+1)	C1 ∨ C2 ∨ C5 ∨ C6					*			X(only)	
16bit ∧ func=add	*(TRO)	C1	*		*	*				X-reg or Y-reg	m7
	*(TRO)	C2			*	*					
	*(TRO)	C3 ∨ C4	*		*						
	*(TRO)	C5	*	*							
	*(TRO)	C1 ∨ C2 ∨ C5 ∨ C6					*			X(only)	
16bit ∧ func=add	/(CS),CS	C1	*		*	*				X-reg or Y-reg	m8
	/(CS),CS	C2			*	*					
	/(CS),CS	C3 ∨ C4	*		*						
	/(CS),CS	C5	*	*							
	/(CS),CS	C1 ∨ C2 ∨ C5 ∨ C6					*			X(only)	
8bit const	<-	C1 ∨ C2 ∨ C5 ∨ C6						*		X-reg	m1
8bit reg-transfer (X<-Y)	<-	C1	*		*	*				X-reg	m10
	<-	C2			*	*					
8bit func=sub	-(+1)	C1	*		*	*				X-reg or Y-reg	m11
	-(+1)	C2			*	*					
8bit func=and	.AND.	C1	*		*	*				X-reg or Y-reg	m12
	.AND.	C2			*	*					
8bit func=or	.OR.	C1	*		*	*				X-reg or Y-reg	m13
	.OR.	C2			*	*					

	ALU operator	Class of X-reg.	Class of Y-reg. C2	C3	C5	C6	KW	KB	KD	Destination register	Micro paradim
16bit ∧ const=0 ∧ X-reg.	<-0	C1∨C2								X-reg	m1
16bit ∧ const=0 ∧ Y-reg.	0->		*		*	*				Y-reg	
16bit ∧ const=1 ∧ func=add1 X-reg.	+1->	C1∨C2								X-reg	m2
16bit ∧ const=1 ∧ func=add1 Y-reg.	+1	V7	*		*	*				Y-reg	m3
16bit ∧ const=(X'KKKK'∨ X'KKK0'∨X'KK0K'∨X'K0KK'∨ X'0KKK')	<-	C1∨C2					*			X-reg	m1
16bit ∧ const ⇒ {byte const(U) / byte const(L)}	<-	C1∨C2						*		X-reg	m1
16bit ∧ reg-transfer (X->Y)	-> -> -> ->	C1 C2 C3∨C4 C5	* * *		* * * 	* * * *				Y-reg	m4
16bit ∧ reg-transfer (X<-Y)	<- <- <- <- <-	C1 C2 C3 C4 C5	* * *		* * * *	* * 				X-reg	m4
16bit ∧ func=add	+ + + +	C1 C2 C3∨C4 C5	* * *	 *	* * * 	* * 				X-reg or Y-reg	m5
	+	C1∨C2∨C5∨C6					*			X(only)	

* : allowable mark

Figure 8. Sample of Table Representation of COSMO-500 Micro-architecture

Resource allocation strategy

We have chosen a resource allocation procedure which is broken into two stages;

(1) register-class allocation:

In this stage of procedure it is decided which class of registers the data variable belongs to.

(2) register-name assignment:

In this stage the name of register for each variable at every program point is decided.

Such a separation of the class allocation from the register assignment aims at the reduction of a backtracking process contained in a resource allocation algorithm. The similar strategy is used in FGS, though the backtracking operation in register assignment procedure is not performed.

Register-class allocation

A classification of set of registers is introduced to deal with an inhomogeneous usage rule of resource in a microcode architecture. Because a register class for a variable is determined depending upon the position and the situations of the expression in which that variable appears, the decision of a class for the variable is performed by examining the situations at every appearance of the variable.

For instance, Figure 9 illustrates a simple register-class allocation problem in the COSMO 500 micro-architecture. This problem has two variables which should be assigned to two registers. Suppose A and B are such variables. From the first form, A can belong to a register class eiter C1 or C2.

```
1  '5' -> A     A ε C1 V C2
2  '8' -> B     B ε C1 V C2
3  A  +  B  ->  B
```

Figure 9. A Simple Register-class Allocation Problem

By examining the second form, B can also belong to a register class C1 or C2. Thus, two variable can be permitted to assign to one of possible four combinations of register class such as

(A, B)=((C1, C1), (C1, C2), (C2, C1), (C2, C2)).

At the examination of the additive operation in the 3rd form, the left operand A is assigned to class C1 and the right operand B is assigned to class C2 by the register usage rule for the additive operation.

But a left and right operands of an additive operation are exchangeable mutually, the final allowable assignment is

(A, B) = (C1, C2) or (A, B) = (C2, C1).

In practical situations, a more sophisticated procedure is required. The class allocation procedure is broken into three stages: an initial stage, an extension stage and an allocation stage.

I. Initial stage: This stage contains an operand analysis and an operator analysis.

(1) An operand analysis is to detect the constant, which is represented internally with the emit field of a micro instruction rather than a register.

(2) An operator analysis is performed to detect the subtractive operator, for the reason why the operands of subtraction are not exchangeable so that allowable class of registers is usually limited.

An operand in a subtraction experssion is called a key variable which is used to decide the undefined class of a variable in the extension stage. The initial and final conditions of the section - the boundary situation of the class assignment in the neighboring sections - are also used to generate a key variable. A class for a key variable is automatically chosen by the system as shown in Figure 10. (a).

II. Extension stage: In this stage, the decision of a register class for appearances of the same variable in the neighboring intermediate-forms, only if the following class-conflict condition is satisfied.

The class-conflict conditions:

(1) The variable should not be a key variable for which the different class is already established.

(2) The candidate class for the variable should belong to the set of allowable class which is previously decided by the register usage rules.

III. Allocation stage:

An usaual micoro-architecture contains the exclusive usage rule of register class, that is, if one variable belongs to certain class, the another variable of the expression should be assigned to the rests of the register class. An allocation stage determins a register class by using this rule as shown in Figure 10 (c). Those variables become to be the new key variable, in turn, and the extension stage is executed again. Thus the extension and allocation stages are repeated alternately until the new key variable can be detected no more. The final result of this procedure for the sample case is given in Figure 10 (d). The symbol * in Figure 10 (d) denotes the unspecified variable for class allocation. This variable is called a selection-point and is used to be a backtracking point by the register-name procedure.

intermediate form	list of variable						
	A	B	C	D	E	F	T
4 -> A	*						
1 -> B		*					
7 -> C			*				
A * D -> T	*			*			*
T + B -> T		*					*
T - C -> E			C1		*		C2
A + C -> F	*		*			*	

(a) an initial stage

intermediate form	list of variable						
	A	B	C	D	E	F	T
4 -> A	*						
1 -> B		*					
7 -> C			*				
A * D -> T	*			*			*
T + B -> T		*					C2
T - C -> E			(C1)		*		(C2)
A + C -> F	*		C1			*	

(b) an extension stage

intermediate form	list of variable						
	A	B	C	D	E	F	T
4 -> A	*						
1 -> B		*					
7 -> C			*				
A * D -> T	*			*			*
T + B -> T		C1	<----------				(C2)
T - C -> E			C1		*		C2
A + C -> F	C2	<---	(C1)			*	

(c) an allocation stage

intermediate form	list of variable						
	A	B	C	D	E	F	T
4 -> A	C2						
1 -> B		C1					
7 -> C			C1				
A * D -> T	C2			C1			C2
T + B -> T		C1					C2
T - C -> E			C1		*		C2
A + C -> F	C2		C1			*	

(d) a result of class assignment

Figure 10. An example of register class assignment

Register-name assignment

The register-class assignment is performend under the assumption that the registers in each class are infinite. On the other hand, the register-name assignment defines a register-name for each variable under the limitation of the number of the given registers.

The register-name assignment algorithm consists of the following steps.

(1) A variable with the class which has an unused registers is assigned to one of these registers.

(2) If there exists no available register, the current status is stored into a stack and the assignment procedure returns back to the last selective point and proceeds the re-assignment of registers.

(3) If every possibility of reassignment is exhausted, a register-release algorithm performs a detection of a variable which is the last to appear from the current program point by referencing to the variable-occurrence table as shown in Figure 11. The register of such a variable is released by transfering its content into the memory.

(4) An automatic garbage collecter for registers is also provided. This works at every program point in order to release a register occupied by the variable of which 'life-cycle' is terminated.

Figure 11 shows an overview of the computation flow of the register-name assignment algorithm.

Figure 12 illustrates a simplified problem for register-name assignment. Assume the given micro-architecture has two register classes and each class consists of 3 registers.
We suppose an intial state is given as shown in Figure 12 (a).

The register-name assignment starts at variable A in the 1st row and assigns A to the register-name R21. The step (1) of the name assignment procedure continues successfully because of the existence of free registers until the variable B in the 5th row (see Figure 12 (b)).
(A variable C in the 3rd row is a selective point and is assigned to register class C1 at the first trial.)

But the all registers in the class C1 is already used at program point of the variable E in the 5th row, so that the procedure enters the backtracking process of the step (2) and restarts from the selective point.

The backtracking process invokes re-assignments of register-name. Figure 12 (c) illustrates the result of register-name assignment.

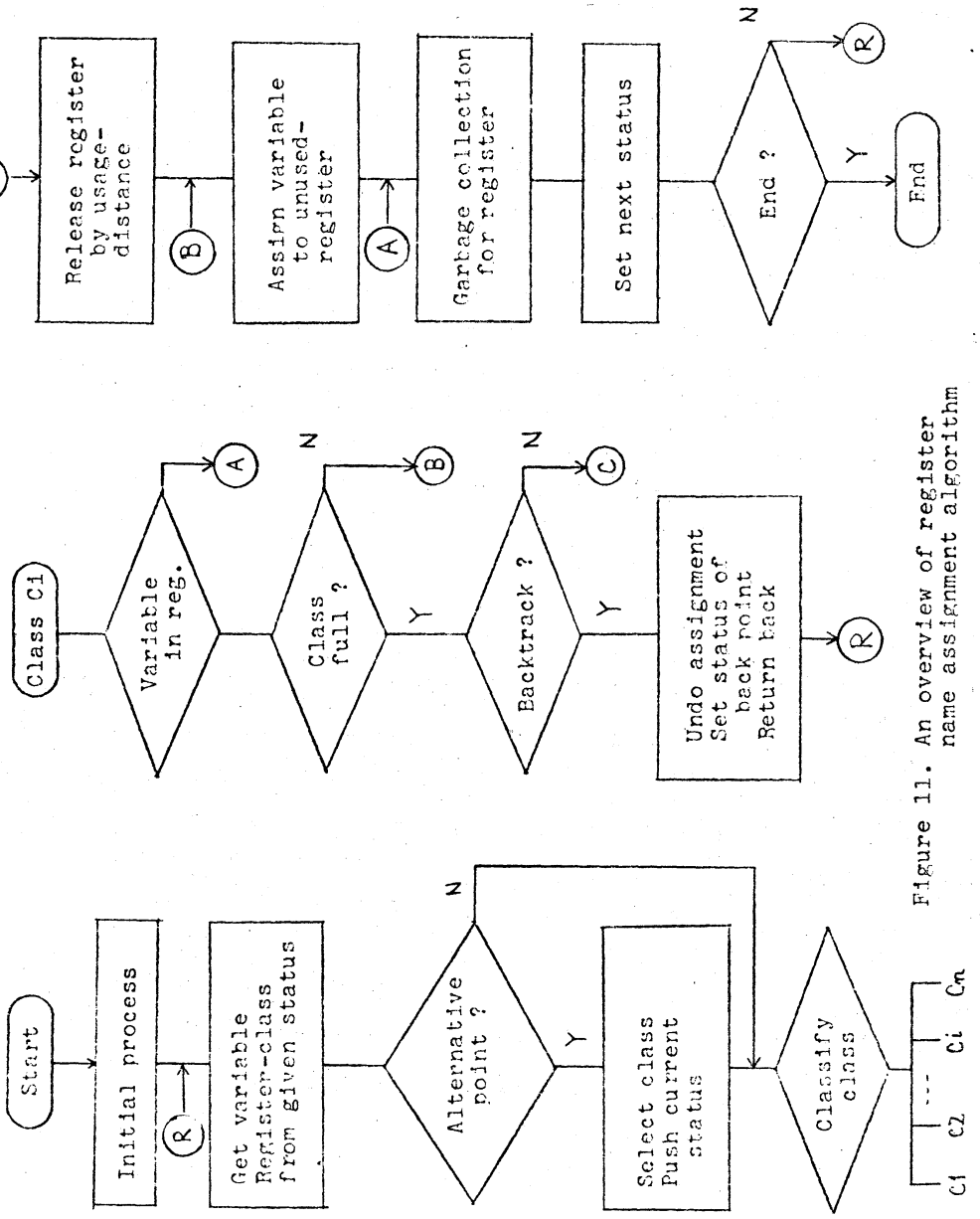

Figure 11. An overview of register name assignment algorithm

intermediate form	register assignment table							variable occurrence table						
	A	B	C	D	E	F	G	A	B	C	D	E	F	G
8 -> A	C2							α	0	0	0	0	0	0
2 -> B		C1						2	α	0	0	0	0	0
6 -> C			X					1	2	α	0	0	0	0
A * D -> G	C2			C1			C2	β	1	4	β	0	0	α
B - F -> E		C1			C1	C2		0	β	3	1	α	β	3
E + F -> D				C1	C1	C2		0	3	2	α	β	β	2

X ∈ (C1 ∨ C2), X is a selective point.

(a) initial status

form-number	List of variables						
	A	B	C	D	E	F	G
(1)	R21						
(2)		R11					
(3)			R12				
(4)	R21			R13			R22
(5)		R11			C1	C2	
(6)				C1	C1	C2	

(b) the first trial (failure)

form-number	List of variables						
	A	B	C	D	E	F	G
(1)	R21						
(2)		R11					
(3)			R22				
(4)	R21			R12			R23
(5)		R11			R13	R21	
(6)				R12	R13	R21	

(c) the result of the second trial (success)

Figure 12. An Example of Register-Name Assignment

Concluding Remarks

An experimental processor which allows the generation of microcode is described. We have completed a construction of a microcode compiler for COSMO 500 computer. The size of this compiler is about 12000 steps in Fortran language.

Although a machine independent version of resource allocation algorithm has not been implemented, it appears to be a promising approach toward a microcode compiler generator.

References

(1) C.J.Tan, "Code optimization techniques for micro-code compilers", NCC 1978, P649-655

(2) K.Mikami and A.Fusaoka, "Compiler construction for a Firmware Generation", IPSJ(Japan), Sep. 1978 (in Japanese)

(3) C.W.Fraser, "A knowledge-based code generator Generator", Proceedings of the Symp. an Artifical Intelligence and Programming Languages, ACM SIGPLAN Notices, vol.12 No 8. Aug. 1977, P126-129

(4) S.L.Graham, "Table Driven Code Generation", Proc. of ACM 78, P476-477.

(5) R.Sethi, "Compiler Register allocation problems", Proc. of 5th annual ACM Symp. on Theory of Computing, P182-195.